Self-Stabilizing Passive Replication for Internet Service Platforms

Koji Hasebe, Kei Yamatozaki, Akiyoshi Sugiki, and Kazuhiko Kato
Graduate School of Systems and Information Engineering
University of Tsukuba
1-1-1 Tennodai, Tsukuba 305-8573, Japan
hasebe@iit.tsukuba.ac.jp, k-yamato@osss.cs.tsukuba.ac.jp, {sugiki,kato}@cs.tsukuba.ac.jp

***Abstract*—We present a self-stabilizing passive replication mechanism for Internet service platforms. The proposed system based on this mechanism aims to provide services despite simultaneous failure of the majority of nodes. To achieve this objective, we use the self-stabilizing consensus algorithm introduced by Dolev et al. [7], instead of a majority-based consensus algorithm such as Paxos. Although our system allows temporary illegal behavior, it eventually converges on the desired state without interruption of services. Thus, the proposed system is useful for providing services that require high availability, but not strict consistency, a prime example of which is an Internet bulletin board for communication during large-scale natural disasters. We implemented a prototype and evaluated it with experiments to demonstrate availability through various patterns of failures.**

***Keywords*-Self-stabilization, passive-replication, Internet service, availability, consensus algorithm**

I. Introduction

With the continuous growth of the Internet, the service availability thereof has become a central issue in today's computing infrastructure. Replication is a widely used technique to provide high availability and fault-tolerance for Internet services. Despite the idea behind this technique being simple, its implementation is complex, because replicating service state on multiple physical nodes requires that each replica remains synchronized and consistent with all others.

To realize consistency of replication, a well-known *passive* replication technique (called *primary-backup* replication) is used [1]. In passive replication, one of the nodes, called the *primary*, processes requests from clients and provides responses. The other nodes, called *backups*, periodically receive state update messages from the primary, allowing them to update their state to match that of the primary. When the primary fails, one of the backups is selected to take over as the new primary. In implementing this technique, a consensus algorithm (cf. [2]) is often used to achieve agreement between all nodes on which node is the current primary, as well as on which update message is the latest. A well-known and widely used consensus algorithm is Paxos [8]. Generally speaking, this algorithm guarantees that $2F + 1$ nodes achieve consensus in an environment where up to F nodes may fail simultaneously. However, if the majority of nodes fail, this algorithm terminates without achieving consensus. Thus, an implementation of passive replication with such a majority-based consensus algorithm cannot continue to provide the services if more than half the system fails as a result of a large-scale disaster.

The objective of this research is to develop an Internet service platform with high availability based on the passive replication technique. In particular, our proposed system aims to provide services despite simultaneous failure of the majority of nodes. To implement such a platform, we use the self-stabilizing consensus algorithm introduced by Dolev et al. [7], instead of a majority-based one such as Paxos. Self-stabilization [5], [6] is a fundamental property of a distributed system that guarantees arriving at a legitimate state in a finite number of steps regardless of the initial state. The main feature of the self-stabilizing consensus algorithm is that it repeatedly invokes one-shot consensus and ensures that every non-failed node eventually agrees on a common value in each one-shot consensus. (In [7], each one-shot consensus is called an *epoch*.) Thus, although the algorithm temporarily allows an inconsistent situation (i.e., nodes may agree on different values at a particular epoch), it eventually reaches a series of agreements even if the majority of nodes fail.

By using this self-stabilizing consensus algorithm, the behavior of passive replication can also be self-stabilized. More precisely, our system continues to provide services as long as at least one node survives. Nevertheless, owing to the use of the self-stabilizing consensus algorithm, our system may display illegal behavior at certain times. Typically, when network delay partitions nodes into separate groups, one node becomes the primary in each group and there may be multiple primaries in the whole system. Thus, although our system cannot be used as the requisite platform to maintain strict consistency as typified by Internet banking services, it is quite useful for Internet services that require high availability, but not strict consistency, such as Internet bulletin boards, especially for communication during large-scale natural disasters.

The rest of this paper is organized as follows. Section 2 discusses related work, while Section 3 describes the proposed self-stabilizing passive replication used for Internet service platforms. Section 4 presents experimental results using our prototype implementation to verify the correctness of our system design and to demonstrate availability with various patterns of failures. Finally, Section 5 concludes the paper and suggests future work.

II. Related Work

To realize consistent replication, various techniques have been proposed and these can be classified into two categories, *active* and *passive* replication, the merits and demerits of which are complementary. In active (or *state-machine*) replication [11], each node processes requests from the clients and transitions independently. Generally, each transition is coordinated across the nodes by means of a consensus algorithm. Although this technique is useful due to its low response time, it has two important drawbacks: high computational resource usage, and the fact that client requests have to be processed in a deterministic manner. On the other hand, passive replication is also useful due to its low computational cost and applicability to nondeterministic services. However, as mentioned in the previous section, a mechanism to agree on the current primary is required in an implementation thereof, which is not necessary in active replication systems.

In recent years, in order to counter these drawbacks, some variants of these schemes have been proposed, such as semi-active replication [12] and semi-passive replication [4]. In [7], the authors proposed self-stabilizing active replication as an application of their consensus algorithm. Compared with active replication based on a majority-based consensus algorithm, their replication technique provides higher availability against simultaneous failure of the majority of nodes. The main motivation of our research is to investigate the applicability of the self-stabilizing consensus algorithm to passive replication. Since the advantages of passive and active replication are complementary, our system is also beneficial when considering its low computation costs and applicability to nondeterministic services.

III. Self-Stabilizing Passive Replication

In this section, we first review the self-stabilizing consensus algorithm introduced in [7], and then describe our proposed self-stabilizing passive replication. Throughout this section, we use the set $N = \{1, 2, \ldots, n\}$ to indicate the set of nodes in a system.

A. Self-Stabilizing Consensus Algorithm

Consensus is a fundamental problem in distributed computing. The problem is usually defined as the task of achieving a goal against failures, where every non-failed node decides on a common output value, starting with different inputs for the nodes in the distributed system. In [7], self-stabilizing consensus is defined as repeated invocations of consensus in the usual sense. More precisely, each one-shot consensus is called an *epoch*, and every node decides on a value for each epoch. Thus, the output of Node i (for $i \in N$) is of the form $\langle v_1^i, v_2^i, \ldots, v_j^i, \ldots \rangle$, where v_j^i is the agreement value for the j-th epoch. The goal is for every node to agree on a common sequence of output values.

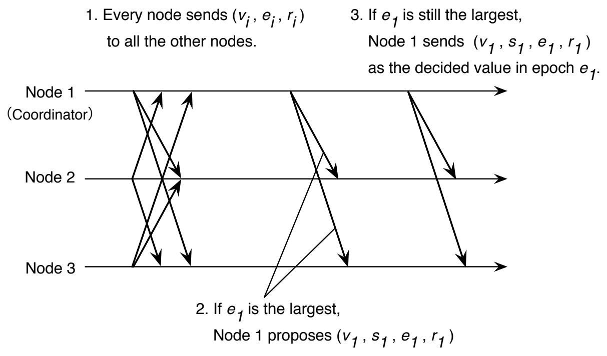
1. Every node sends (v_i, e_i, r_i) to all the other nodes.
3. If e_1 is still the largest, Node 1 sends (v_1, s_1, e_1, r_1) as the decided value in epoch e_1.
Node 1 (Coordinator)
Node 2
Node 3
2. If e_1 is the largest, Node 1 proposes (v_1, s_1, e_1, r_1)

Figure 1. Process of self-stabilizing consensus

To solve this problem, the self-stabilizing consensus algorithm is based on a method using a rotating coordinator. In this algorithm, Node i has a pair of variables $\langle e_i, r_i \rangle$ for each $i \in N$. Intuitively, e_i indicates the current epoch, the output of which is in the process of being determined by the node, while r_i, referred to as *round*, indicates the number of trials in the particular epoch. In terms of these variables, the algorithm achieves self-stabilizing consensus by the following steps. (The process is illustrated in Fig. 1, where Node 1 is the coordinator.)

1) Every node decides a proposal v_i and sends tuple $\langle v_i, e_i, r_i \rangle$ to the other nodes. Next, if its own node ID (i.e., the value i) is equal to $r \ mod \ n$, the node becomes the *coordinator*.
2) The coordinator compares its own epoch number e_i with the latest epoch numbers sent by the other nodes. If its own epoch number is the highest (more precisely, $e_i > e_j$ or $e_i = e_j$ with $r_i > r_j$ for any $j \neq i$), the coordinator sends tuple $\langle v_i, s_i, e_i, r_i \rangle$ to all the other nodes as the proposal. Here, s_i (referred to as the *state*) is the sequence of values decided thus far. Otherwise, the coordinator increases r_i by 1, and then returns to Step 1.
3) If Node i is still the coordinator, it again compares its own epoch number e_i with the latest epoch numbers sent by the other nodes. If e_i is still the largest, Node i sends tuple $\langle v_i, s_i, e_i, r_i \rangle$ to all the other nodes as the decided value, and sets the epoch and round numbers to $e_i + 1$ and 0, respectively. Otherwise, Node i increases r_i by 1 and returns to Step 1. Each of the other nodes updates its state with s_i when receiving the decided value.

This algorithm ensures the following properties (cf. [7] for the detailed definitions and proofs):

- *eventual termination*: each non-crashed node eventually decides on a value for every epoch,
- *eventual validity*: each non-crashed node eventually

decides on the initial value of some non-crashed node in every epoch, and

- *eventual agreement*: no two non-crashed nodes decide on different values for any epoch,

provided that

- fail-stop failure of any node can occur and a maximum of $n-1$ of the n nodes fail at the same time,
- the network is reliable, i.e., any sent message eventually arrives at its destination and message order is ensured, and
- there exists a self-stabilizing $\Diamond S$ failure detector that satisfies *strong completeness* and *eventual weak accuracy*.

(Throughout this paper, we assume the same failure model.)

Here, we would like to clarify the differences between Paxos and this algorithm. Unlike Paxos, the self-stabilizing consensus algorithm does not terminate even if the majority of nodes fails. On the other hand, this algorithm does not guarantee that every node *always* agrees on a common sequence of values. In other words, a situation may occur in which some nodes decide on different values in a particular epoch. However, such different values eventually converge on a common one. As we shall see in the next subsection, this feature of self-stabilization is inherited by our proposed system.

B. System Design

Overview. Fig. 2 gives an overview of our platform. The proposed platform consists of a number of nodes (i.e., physical machines). In the same manner as the usual passive replication, one specific node, called the primary, provides services, while the other nodes, called backups, periodically receive state update messages from the primary. To encapsulate the service state, we use a virtual machine (VM) technique, which enables the service state to be checkpointed and migrated to another node. The service state can be complete, including everything from the operating system to the service software, without modifying the service code. This technique is widely used to implement active and passive replication, with VM-FIT [10] and Remus [3] being prime examples.

To realize self-stabilizing passive replication, our system provides the mechanisms for state update and replacement of a new primary based on the self-stabilizing consensus algorithm.

State update. Fig. 3 illustrates the process of state update. The detailed steps are as follows.

1) The primary periodically generates an update message of its current state and sends the message accompanied by its unique version number (denoted by v) to all the backups.

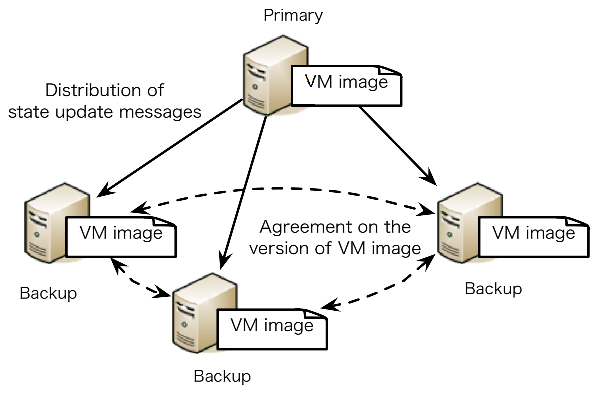
Primary
VM image
Distribution of state update messages
VM image
Agreement on the version of VM image
VM image
Backup
VM image
Backup
Backup

Figure 2. Overview of the system architecture

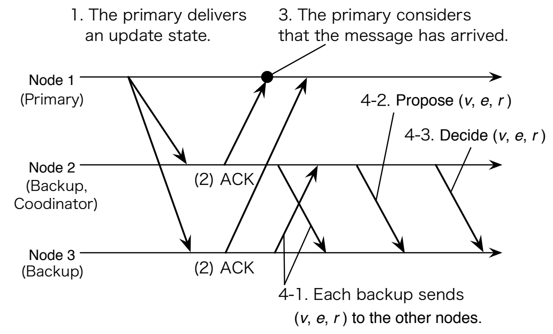
1. The primary delivers an update state.
3. The primary considers that the message has arrived.
Node 1 (Primary)
4-2. Propose (v, e, r)
4-3. Decide (v, e, r)
Node 2 (Backup, Coodinator)
(2) ACK
Node 3 (Backup)
(2) ACK
4-1. Each backup sends (v, e, r) to the other nodes.

Figure 3. Process of state update

2) Each backup responds with an acknowledgement to the primary when receiving the update message and proceeds to Step 4 below.
3) If the primary receives an acknowledgement from at least one backup, it proceeds to Step 1 and generates the next update message. Otherwise (i.e., it has not received any acknowledgements after a certain period of time), the primary resends the same update message.
4) After receiving an update message, the backup executes the self-stabilizing consensus algorithm. That is, (4-1) the backup sets both its own proposing value v_i and epoch number e_i to v, as well as its own round number r_i to 0, and then sends tuple $\langle v_i, e_i, r_i \rangle$ to all the other backups; (4-2) if the backup becomes a coordinator, it proposes tuple $\langle v_i, e_i, r_i \rangle$; (4-3) if its epoch number is still the largest, the coordinator decides the update message of version v for epoch e_i and sends tuple $\langle v_i, s_i, e_i, r_i \rangle$ to all the other backups, and then increments its own epoch number by 1 and returns to Step 2. On the other hand, when a backup that is not the coordinator receives the coordinator's decision $\langle v_i, s_i, e_i, r_i \rangle$, it sets its own epoch and round numbers to $e_i + 1$ and 0, respectively, as well as its state to s_i, and then returns to Step 2.

Replacement of the primary. In addition to the above process, all nodes continually execute the self-stabilizing consensus algorithm in parallel to agree on the current primary. This can be realized by regarding the agreement value (i.e., v_i) as the ID number of the current primary. That is, each node initially sets this value to 1 and only changes it when it realizes that the primary has failed. Otherwise, it sets the agreement value to its own ID number. According to this algorithm, at least one node can become the new primary.

Remarks. By these mechanisms, our system behaves in a self-stabilizing manner in the following sense. In the proposed system, there is at least one primary and services can be provided continuously as long as one node survives. Moreover, the system can tolerate various patterns of simultaneous failures. We demonstrate this through various experiments in the next section. On the other hand, when network delay partitions nodes into different groups, multiple nodes may become the primary. However, this illegal behavior does converge on the desired state, i.e., one of the nodes remains the primary and the others become backups. Thus, although our system cannot be applied to services that require strict consistency, it is quite useful for services that require high availability. For example, in the case of Internet bulletin boards, our system can recover from any illegal situation by merely merging the individual texts stored in the multiple primaries.

Our system does have some limitations, the most critical of which is that the system may loose the latest state when network delay occurs among backups. This problem is observed in the fifth experiment in the next section. We have not addressed this problem in this paper; instead it is left as one of our future works.

IV. Experiments

To verify the correctness of our algorithm presented in the previous section and to demonstrate the availability with various types of node and network failures, we conducted experiments with our current prototype implementation of the proposed system. Our prototype consists mainly of four modules: the failure detectors, the consensus algorithm, the primary election, and the passive-replication mechanism, with the first two based on the suite of self-stabilizing consensus algorithms in [7]. These modules behave independently and communicate with one another by means of asynchronous message passing. To implement these concurrent processes and to optimize their productivity, our prototype was implemented in Scala [9] and consists of 1250 lines of code. In addition to the Scala Actors library, to manage VMs we also used various APIs provided by Kumoi [13], a scripting environment for managing collective VMs in large-scale data centers.

A. Experimental Setup

The experiments were conducted on 5 identical PC servers, each of which was equipped with Dual Xeon 3.60 GHz CPUs, 2 GB memory, and a single 36 GB SCSI disk. Each server was connected to a single switch via a 1000Base-T network adapter. CentOS 5.5 and 5.4 were used for host and guest OSs, respectively. We used the Xen 3.03 virtual machine monitor, and the “xm” command to save the current state of a virtual machine and write it to a disk file as a VM image. VM images were transferred using Secure Copy (SCP). Implementation of the self-stabilizing failure detector includes a heartbeat mechanism. By means of the failure detector, each backup considers the current primary to have failed if no heartbeat is received for 10 seconds.

For each experiment, all the machines (called Nodes 1–5) were initially booted at the same time and Node 1 was selected as the primary server. Then, we generated a failure and observed the system behavior for 600 seconds. Failures of the machines and network were generated artificially by switching off the machines and thus, cutting off the communication lines. As examples of possible failures in real systems, we considered the following six cases.

1) **Serial failures:** The primary (Node 1) fails after sending the current snapshot, and then the next primary also fails in the same way.
2) **Failure of the majority of nodes:** Three machines (Nodes 1–3), including the primary, fail at the same time.
3) **Network partitioning:** Network partitioning occurs, splitting the nodes into two groups, Nodes 1–2 and 3–5, and then the system recovers.
4) **Failure during election:** The primary (Node 1) fails, and then the candidate for the next primary also fails whilst in the *Decide* state (i.e., Step 3 of the consensus algorithm).
5) **Loss of the latest snapshot:** The primary (Node 1) and a backup (Node 2), which has the latest snapshot, fail at the same time. (In this case, Nodes 3–5 have not yet received the latest snapshot.)
6) **Booting a backup with an old version of the snapshot:** First, Version 1 of the snapshot sent by the primary (Node 1) arrives at all backups (Nodes 2–5). Then, due to network delay, the next snapshot, i.e. Version 2 arrives only at Nodes 4 and 5, and not at Nodes 2 and 3. Nevertheless, Nodes 2 and 3 recognize that Node 1 is alive. (Note that this situation is possible because transfer of a snapshot takes some time, whereas the heartbeat arrives successfully at regular intervals.) After this, the primary fails and Node 2 is elected as the next primary.

B. Experimental Results

The results obtained from these experiments are given below. For each case, we also present bar charts depicting

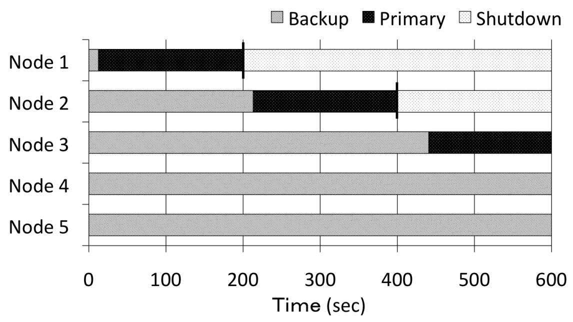

Figure 4. Case 1: Serial failures

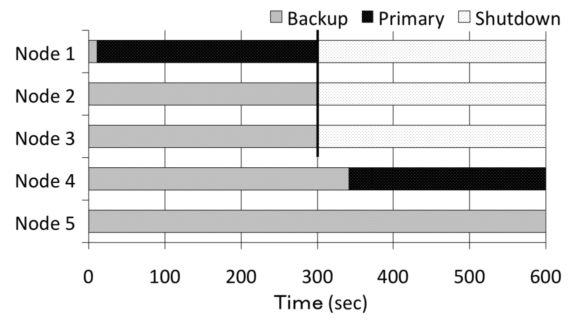

Figure 5. Case 2: Failure of the majority of nodes

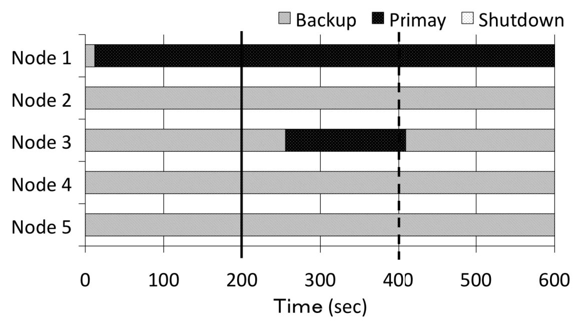

Figure 6. Case 3: Network partitioning

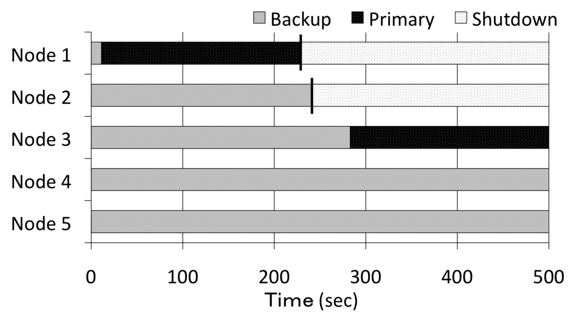

Figure 7. Case 4: Failure during election

the changes in state in each node during the experiments, with the heavy vertical lines indicating the points at which failures occur.

1) Serial failures: Fig. 4 illustrates the behavior of the system when two failures of the primary occur after a lapse of 200 and 400 seconds. In this experiment, our system takes 13 and 40 seconds, respectively, to recover the service (i.e., to complete the replacement of the primary) by means of the latest version of the snapshot. (Here, the first recovery is faster than the second, because Node 1 does not deliver the snapshot before the first failure.) This result indicates that our system can recover from a single node failure within at most one minute.

2) Failure of the majority of nodes: Fig. 5 illustrates the behavior of the system when a failure occurs after a lapse of 300 seconds. In this experiment, Node 4 becomes the primary after 41 seconds. This result indicates that our system has the ability to recover from failure of the majority of nodes. This cannot be achieved if a majority consensus algorithm is used for the primary election.

3) Network partitioning: Fig. 6 illustrates the behavior of the system when a failure occurs and the system recovers after a lapse of 200 and 400 seconds, respectively. After network partitioning, Nodes 3–5 are not able to receive a heartbeat from the primary, and therefore, in this group, Node 3 is elected as the primary and starts the service after 55 seconds. After recovering from the failure, Node 3 recognizes that Node 1 is still alive, having received a heartbeat from Node 1. Node 1 also recognizes that Node 3 is acting as the primary. Then, according to their epoch and round numbers, Node 1 is elected as the primary and Node 3 becomes a backup and terminates its VM after a lapse of 410 seconds. This result shows that our system allows temporary illegal behavior, but eventually converges on a legal state after network communication recovers.

4) Failure during election: Figure 7 illustrates the behavior of the system when a failure occurs during an election process. In this experiment, the primary (Node 1) is stopped, and then, after 12 seconds, the candidate for the next primary (Node 2) is also stopped while in the *Decide* state. Subsequently, Nodes 3–5 recognize that Node 2 has failed, and thus, these nodes elect Node 3 as the next primary and it begins to provide the service 42 seconds after the failure of Node 2. This result indicates that our system can recover from failure of a single node regardless of its state.

5) Loss of the latest snapshot: Fig. 8 illustrates the behavior of our system when the primary and a backup, which has the latest snapshot, fail simultaneously. In this experiment, after the simultaneous failure of the primary (Node 1) and backup (Node 2) with the latest snapshot, Node 3 is elected as the next primary and starts to provide the service with an older version of the snapshot. This takes 35 seconds. As mentioned in the previous section, this result shows that our system cannot guarantee that the latest state of the primary

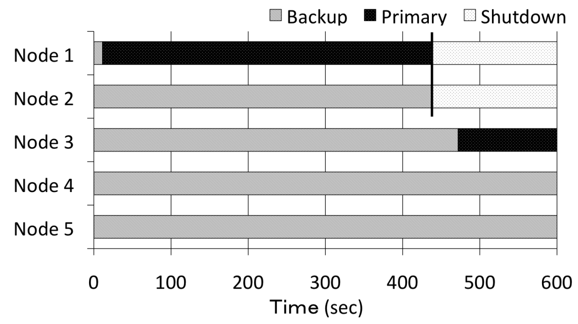

Figure 8. Case 5: Loss of the latest snapshot

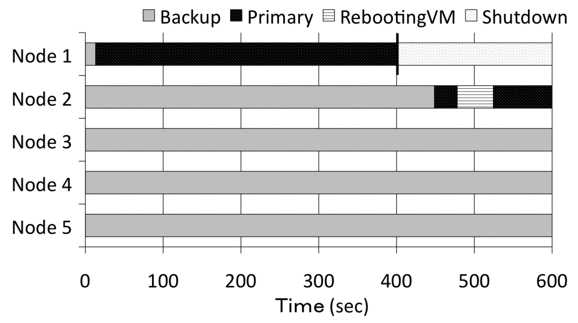

Figure 9. Case 6: Booting a backup with an old version of the snapshot

is always inherited by the next primary.

6) Booting a backup with an old version of the snapshot: Fig. 9 illustrates the behavior of our system when a failure occurs after a lapse of 400 seconds. After the primary (Node 1) is stopped, Node 2 is elected as the primary and starts the service with Version 1 of the snapshot after a lapse of 448 seconds. During this process, Nodes 4 and 5 successfully receive Version 2 of the snapshot and continue to behave as backups. Then, when Node 2 realizes that Nodes 4 and 5 have a later version of the snapshot, Node 2 stops its own VM after a lapse of 478 seconds, and then receives the later one from Node 4. After a lapse of 524 seconds, Node 2 completes rebooting its VM with the received snapshot and begins providing the service.

V. Conclusions and Future Work

We have proposed self-stabilizing passive replication for Internet service platforms. To achieve our objective, we use a self-stabilizing consensus algorithm, instead of a traditional majority-based one. As shown by the results of our experiments, our system does indeed behave in a self-stabilizing manner, and can tolerate various patterns of failures, including simultaneous failure of the majority of nodes. The system is useful for providing Internet services requiring high availability as typified by Internet bulletin boards for communication during large-scale natural disasters.

Regarding future work, one of the most interesting and worthwhile directions is to improve some of the problems in our replication mechanism, especially the problem whereby the latest state may be lost due to a specific pattern of failures. We intend to address this problem by using a more sophisticated technique, such as superstabilization [6], which allows to combine benefits of both self-stabilization and dynamic algorithms.

Acknowledgments

This study is partially supported through the SCOPE program by the Ministry of Internal Affairs and Communications of Japan. This study is also supported in part by the Grant-in-Aid for the Global COE Program on "Cybernics: fusion of human, machine, and information systems" at the University of Tsukuba.

References

[1] N. Budhiraja, K. Marzullo, F. B. Schneider, S. Toueg. The primary-backup approach, *Distributed Systems (2nd ed.)*, ACM press/Addison-Wesley Publishing Co., pp. 199-216, 1993.

[2] G. Coulouris, J. Dollimore, T. Kindberg. Chapter 12 in *Distributed Systems: Concepts and Design (4th Edition)*, Addison Wesley, 2005.

[3] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication, *USENIX NSDI'08*, pp. 161-174, 2008.

[4] X. Defago and A. Schiper. Semi-Passive Replication and Lazy Consensus, *Journal of Parallel and Distributed Systems*, vol. 64 (12), pp. 1380-1398, 2004.

[5] E. W. Dijkstra. Self-Stabilizing System in Spite of Distributed Control, *Communication of the ACM*, vol. 17 (11), pp. 643-644, 1974.

[6] S. Dolev. *Self-Stabilization*, MIT Press, 2000.

[7] S. Dolev, R. I. Kat, and E. M. Schiller. When Consensus Meets Self-Stabilization, *Journal of Computer and System Science*, vol. 76 (8), pp. 884-900, 2010.

[8] L. Lamport. The Part-time Parliament, *ACM TOCS*, vol. 16 (2), pp. 133-169, 1998.

[9] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*, Artima, 2008.

[10] H. P. Reiser and R. Kapitza. Hypervisor-Based Efficient Proactive Recovery, *IEEE SRDS'07*, pp. 83-92, 2007.

[11] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, *ACM Computing Surveys*, vol. 22 (4), pp. 299-319, 1990.

[12] D. Stodden. Semi-Active Workload Replication and Live Migration with Paravirtual Machines, *Xen Summit, Spring 2007*, 2007.

[13] A. Sugiki, K. Kato, Y. Ishii, H. Taniguchi, and N. Hirooka. Kumoi: A High-Level Scripting Environments for Collective Virtual Machines. *16th International Conference on Parallel and Distributed Systems*, 8 pages, 2010 (to appear).